Utilizing Large Language Models to Analyze PSR.exe Recorded Input for Computer Use

YUAN Tianyu

JAN 13, 2025

Abstract

The rapid advancement of Large Language Models (LLMs) has opened new frontiers in automating complex workflows. This paper explores an innovative approach to computer use simulation by leveraging Large Language Models (LLMs) to parse and interpret data recorded by PSR.exe, a tool designed to capture user's mouse and keyboard operations. We propose a method to extract, analyze, and replicate user interactions recorded in MHT files. By decoding screenshots and extracting action sequences, we aim to develop an automated process that enables applications to emulate user operations effectively. The workflow combines BeautifulSoup for XML parsing, base64 for image decoding, and LLMs for semantic analysis. Results show that our method is lightweight, versatile, and capable of ensuring precision and adaptability while reducing dependency on external tracking tools.

1 Introduction

1.1 Background and Motivation

The Problem Steps Recorder (PSR.exe) records user interactions with computer interfaces, generating a detailed step-by-step MHT file containing screenshots and action descriptions. However, these files are inherently static and lack the operational structure required for automated simulation. Large Language Models (LLMs) such as GPT-4 offer a powerful mechanism for understanding and transforming such static records into actionable workflows, paving the way for automation in computer use.

1.2 Research Objectives

This paper outlines an LLM-driven framework for parsing MHT files created by PSR.exe, with the following objectives:

1. Extract mouse and keyboard interactions from MHT files.

2. Generate step-by-step actionable sequences for computer use simulation.
3. Incorporate screenshots for improved accuracy and determinism.
4. Visualize the final workflows through mermaid-based flowcharts.

1.3 Significance

Our approach offers a lightweight solution for automating computer operations without relying on resource-intensive tracking tools. By integrating LLMs with PSR.exe, we enable a streamlined and generalized process for analyzing and reproducing user actions across diverse applications.

2 Related Work

2.1 Interface Automation

Past studies in interface automation have primarily relied on screen-tracking tools or APIs such as Win32. These solutions, while effective, impose significant performance overhead on host systems.

2.2 LLMs in Workflow Generation

LLMs have demonstrated proficiency in natural language understanding and code generation, making them suitable for parsing semi-structured data and generating executable workflows. Notable applications include robotic process automation and user interface testing.

2.3 PSR.exe Utilization

PSR.exe has been widely used for troubleshooting and documentation. However, its utility in automation remains underexplored, particularly in conjunction with modern AI technologies.

3 Methodology

3.1 Data Extraction

3.1.1 Parsing MHT Files

The recorded MHT file is parsed using Python libraries such as BeautifulSoup. The file contains XML-like structures that encapsulate action descriptions, screenshots, and metadata. Steps include:

- Extracting XML units from the HTML body.
- Isolating action descriptions and associated screenshots.

3.1.2 Screenshot Decoding

Screenshots embedded in the MHT file are Base64-encoded JPEGs. Decoding is performed using Python's base64 and PIL libraries. Extracted images provide visual context for each action.

3.2 Step Refinement

3.2.1 Structuring Steps into JSON

Each parsed step is formatted into JSON, containing the following fields:

```
{
  "step": 1,
  "description": "Open File Menu",
  "screenshot": "base64_encoded_image",
  "action": "click"
}
```

3.2.2 Incorporating LLMs

Steps and screenshots are transmitted in batches to an LLM. The LLM processes the input, generating actionable descriptions in the format: "Application - Field - Action". Examples include:

- **Notepad - File Menu - Click**
- **Browser - URL Bar - Type**

3.3 Workflow Optimization

3.3.1 Eliminating Redundant Actions

Redundant steps such as unnecessary mouse scrolling are identified and removed. Actions involving dropdown menus are flagged for conditional checks.

3.3.2 Workflow Summarization

Steps are grouped by application context to form cohesive workflows. This segmentation aids in the logical organization of tasks.

3.4 Visualization

The finalized workflow is converted into a mermaid-compatible format and rendered as an HTML flowchart. The workflow depicts sequential and conditional branches, aiding in intuitive understanding.

4 Implementation

4.1 Tools and Libraries

- **Python**: Base64 decoding, HTML parsing (BeautifulSoup).
- **Pillow**: Image processing.
- **Mermaid.js**: Workflow visualization.
- **LLM**: GPT-4 or equivalent for step analysis.

4.2 Example Code

4.2.1 Parsing MHT Files

```
from bs4 import BeautifulSoup
import base64

# Parse MHT for steps
def parse_mht(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        soup = BeautifulSoup(file.read(), 'html.parser')
    return soup
```

4.2.2 Decoding Screenshots

```
from PIL import Image
from io import BytesIO

def decode_image(base64_str):
    image_data = base64.b64decode(base64_str)
    return Image.open(BytesIO(image_data))
```

4.2.3 Sending Data to LLM

```
def process_step_with_llm(step):
    llm_input = {
        "description": step['description'],
        "screenshot": step['screenshot']
    }
    return llm_model.generate(llm_input)
```

5 Results

5.1 Advantages

- **Low System Overhead:** Operates without additional tracking tools.
- **High Accuracy:** Integrates screenshots for precise context.
- **Automation-Friendly:** Outputs standardized action sequences and workflows.
- **Actionable Sequences:** Extracted sequences are concise and structured, with enhanced clarity from screenshots.
- **Workflow Visualization:** Generated flowcharts accurately represent user operations, facilitating automated replication.
- **Performance Metrics:** Our approach exhibits low resource overhead, making it suitable for real-time applications.

5.2 Limitations

- **Dependence on Screenshot Quality:** Poor image quality may impact LLM interpretation.
- **Model Limitations:** LLMs require fine-tuning for domain-specific tasks.

5.3 Discussion

The proposed methodology successfully automates the parsing and interpretation of PSR records. However, challenges such as ambiguous descriptions or missing screenshots require further enhancements, such as integrating heuristic algorithms or advanced image recognition models.

5.4 Future Work

- **Enhanced Image Analysis:** Use vision models for deeper analysis of screenshots.
- **Adaptive Learning:** Train LLMs with domain-specific data.
- **User Interaction:** Develop interfaces for user feedback during workflow generation.

6 Conclusion

This paper presents a novel framework integrating PSR.exe and LLMs for computer use simulation. By parsing MHT files, refining steps, and visualizing workflows, our methodology enables efficient automation without additional tracking tools. Future work will explore adaptive improvements, including real-time feedback loops and expanded application domains.

7 References

1. https://github.com/u3588064/WorkflowLearner
2. Brown, T. et al. "Language Models are Few-Shot Learners." *NeurIPS*, 2020.
3. Microsoft. "Using Problem Steps Recorder." *Microsoft Support*, 2023.
4. BeautifulSoup Library for Parsing MHT Files
5. Vaswani, A. et al. "Attention Is All You Need." *NeurIPS*, 2017.


```
import base64
import json
import email
from email import policy
from email.parser import BytesParser
from bs4 import BeautifulSoup
from PIL import Image
from io import BytesIO

# Parse MHT file for images
def parse_mht_for_images(file_path):
    with open(file_path, 'rb') as file:
        msg = BytesParser(policy=policy.default).parse(file)

    images = {}
    for part in msg.iter_parts():
        if part.get_content_type() == 'image/jpeg':
            content_location = part.get('Content-Location')
            base64_data = part.get_payload(decode=False)
            images[content_location] = base64_data

    return images

# Parse MHT file for steps
def parse_mht_for_steps(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        content = file.read()
    soup = BeautifulSoup(content, 'html.parser')
```

```
    xml_data = soup.find('script', {'id': 'myXML'}).string
    xml_soup = BeautifulSoup(xml_data, 'xml')

    steps = []
    for action in xml_soup.find_all('EachAction'):
        step_data = {
            'step': action.get('ActionNumber'),
            'description': action.find('Description').text,
            'screenshot': action.find('ScreenshotFileName').text,
            'action': action.find('Action').text
        }
        steps.append(step_data)
    return steps

# Decode Base64 image
def decode_base64_image(base64_data):
    image_data = base64.b64decode(base64_data)
    image = Image.open(BytesIO(image_data))
    return image

# Process steps with LLM
def process_steps_with_llm(steps, images):
    results = []
    for step in steps:
        if step['screenshot'] in images:
            base64_data = images[step['screenshot']]
            screenshot = decode_base64_image(base64_data)
            screenshot.save('/work/' + step['screenshot'], format='JPEG')
            # Mock LLM processing function
            llm_output = process_with_llm(step,
                screenshot='/work/' + step['screenshot'])
            step['llm_output'] = llm_output
            results.append(step)
    return results

# Organize steps
def organize_steps(results):
    organized_steps = []
    for result in results:
        if result not in organized_steps:
            organized_steps.append(result)
    return organized_steps

# Summarize workflow
def summarize_workflow(organized_steps):
    workflow = {}
    for step in organized_steps:
        app = step['description'].split('-')[0]
        if app not in workflow:
```

```
            workflow[app] = []
        workflow[app].append(step)
    return workflow

# Main function
def main(mht_file_path):
    steps = parse_mht_for_steps(mht_file_path)
    images = parse_mht_for_images(mht_file_path)

    results = process_steps_with_llm(steps, images)
    organized_steps = organize_steps(results)
    workflow = summarize_workflow(organized_steps)

    with open('results.json', 'w', encoding='utf-8') as file:
        json.dump(results, file, ensure_ascii=False, indent=4)

    print("Final␣Action␣Sequence␣and␣Workflow:")
    print(json.dumps(workflow, ensure_ascii=False, indent=4))

if __name__ == "__main__":
    mht_file_path = 'Recording_20241120_2136.mht'
    main(mht_file_path)
```